AVL Tree Performance Report

Siyamthanda Phakathi

Submitted to the University of Cape Town
In accordance to the institutions Academic
Dishonesty Policy

Faculty of Science

CSC 2001F

Introduction

Various AI systems use general knowledge to answer questions and use reasoning. These large language models have to be trained and learning from web text may be inaccurate. To better train these models, researchers have curated verified datasets that contain generic statements about the world. One of these datasets is known as GenericsKB, which contains over 3 million verified general statements about the world.

Problem Statement

In this experiment I will be required to populate an AVL tree with entries from GenericsKB. Using instrumentation I will measure the time taken for insertions and searching as the dataset size n increases.

Object Oriented Design

For this assignment I created 4 java classes and 1 python script. These in conjunction with the classes provided by Professor Hussein Suleman allowed me to fulfil the needs of this project. Below are the classes I created and their functionality.

Entry.java is an entry object where the term, statement and confidence score are all properties of the object. It consists of basic getters and setters and a compareTo() method to compare entries by their term. It also has a basic toString() method.

The AVLTree has been constructed using the classes provided by professor Hussein Suleman.

KnowledgeBaseAVL.java is the backend for all tree manipulation and txt processing. Two AVL trees, namely entries and queries, are initialized.

- AddEntries(String fileName): Takes in the name of a txt file such as GenericsKB.txt and tokenizes the term, statement and confidence score. It then used those values to create a new Entry object and add it to the entries tree. This function is called assuming the tree is still empty.
- AddQueries(String fileName): Reads in a txt file full of terms to search for into the queries tree.
- appendEntries(String fileName): Adds new entries to a populated entries tree. insertCounts for entries is reset to 0 as to count the time taken to add new entries to a tree. Variables insertCounts and findCounts are initialized upon creation in BinaryTree class and are set to 0. They are incremented everytime there is a comparison by term for the insert and find function in AVLTree.
- Search(String query): Searches for a term in the tree. If found it is printed out otherwise term not found is printed out.
- SearchAll() recursively searches for all terms in the queries tree one by one. Printing out if it is found or not. It starts searching from the root node
- getInsertOperationsCount() returns the number of insert operations done so far.
- getSearchOperationCount() returns the number of search operatins done so far.

AVLTreeTest.java is used for preliminary testing ensuring the knowledge base can be populated and insert and search functions work as intended. It takes in a premade file of 10 entries (EntriesTest.txt) and 10 queries (QueriesTest.txt) and tests all the functions of the knowledge base and prints out the results to ResultsEntriesTest.txt. You can run it from the terminal using **make test**

GenericsKbAVLApp.java takes in a randomised data file of size n and loads it to our entries tree through the knowledge base. It then takes in a randomised query file and a randomised inserts file of size m. We will first search for the queries and note the search time and then append the tree with our inserts file. This will measure the time taken to insert m entries into a tree of size n. Output is printed to a Results text file. To run manually with your own files use the following command substituting the variables for the actual file names:

java -cp bin GenericsKbAVLApp dataFile queryFile insertsFile

Otherwise you can run it automatically using the provided files by using

make run

GenericsKBAVLTest.py is used to automate the entire experiment portion including creating random subsets of the data across multiple runs for more accurate data collection. That data is used to plot a graph for search and insert time using **matplotlib**. Details of the experiments are in the next section.

Experiment

In this experiment we are testing the time complexities for insertions and searches in an AVL tree. We want to see how varying the size of the dataset n will effect our numbers of comparison. To do so we will use 10 values of n, spaced equally apart logarithmically. **One** run of our testing process will go as follows:

- Create a subset of m entries that are not in GenericsKB.txt to be inserted called Entries{m}.
- Create a subset of m queries from GenericsKB-queries.txt to be searched called Queries{m}.
- These created files will be used for all values of n as to measure the effect of changing our dataset not our insertions and queries.
- Create a subset of n entries from GenericsKB.txt called Generics{n}.txt for all values of n.
- Run GenericsKbAVLApp for all our n entries files (Generics{n}.txt) and for each we will use the same Entries and Queries subset to measure our time.
- Determine the best case, worst case and average of all the values we receive from this.

We will then repeat the process above r times to ensure our numbers are more accurate over time. Once all these values have been collected plot a graph of all these values and compare with our theoretical complexity for insert and search.

All of this can be done manually but I have used a python script to automate the entire process. The code for the automated process will be explained in the creativity section below are the results and conclusions from the experiment.

Testing Part 1

For part 1 I used AVLTreeTest.java and used the following 10 queries and 10 entries. To run the test of loading and finding queries run **make test** in the terminal to test the loading of files and our instrumentation.

TestEntries.txt

```
criminologist   Criminologists are workers. 1.0
albatross   Albatrosses have (part) faces.  1.0
isolated organ  Isolated organs are rich in endothelial cells, vascular smooth muscle and c
posterior molar Posterior molars erupt at the back of the row and slowly move forward.  0.7
distinct structure  Distinct structures serve functions.    1.0
melatonin   Melatonin is produced by glands.    1.0
packer  Packers have (part) arms.   1.0
running Running  car crashs.    1.0
chatterbox  A chatterbox is a helleborine   1.0
structural abnormality  Structural abnormalities involve changes in the structure of one or
```

TestQueries.txt

```
criminologist
albatross
isolated organ
posterior molar
distinct structure
appliances
Swamp Izzo
bad hair
Yves Saint Lauren
magnolia
```

The results of the test are as follows:

ResultsTestEntries.txt

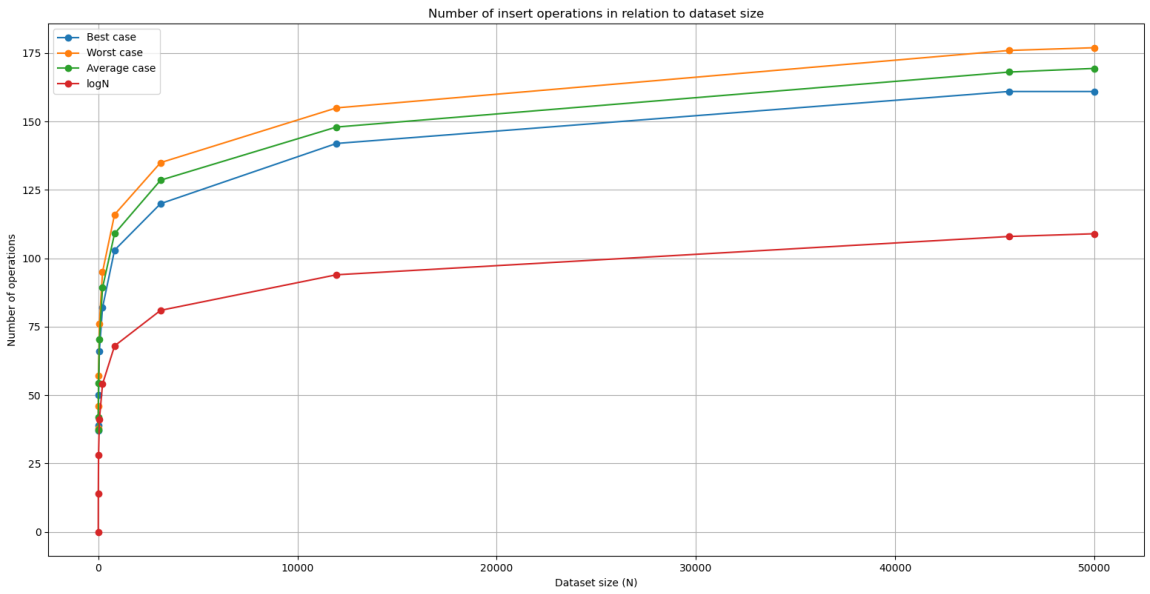
```
criminologist: Criminologists are workers.(1.0)
albatross: Albatrosses have (part) faces.(1.0)
Term not found:Swamp Izzo
Term not found:Yves Saint Lauren
Term not found:appliances
Term not found:bad hair
isolated organ: Isolated organs are rich in endothelial cells, vascular smooth muscle and other types of ce
distinct structure: Distinct structures serve functions.(1.0)
posterior molar: Posterior molars erupt at the back of the row and slowly move forward.(0.719)
Term not found:magnolia
Number of operations for insertions: 32
Number of operations for search: 30
```

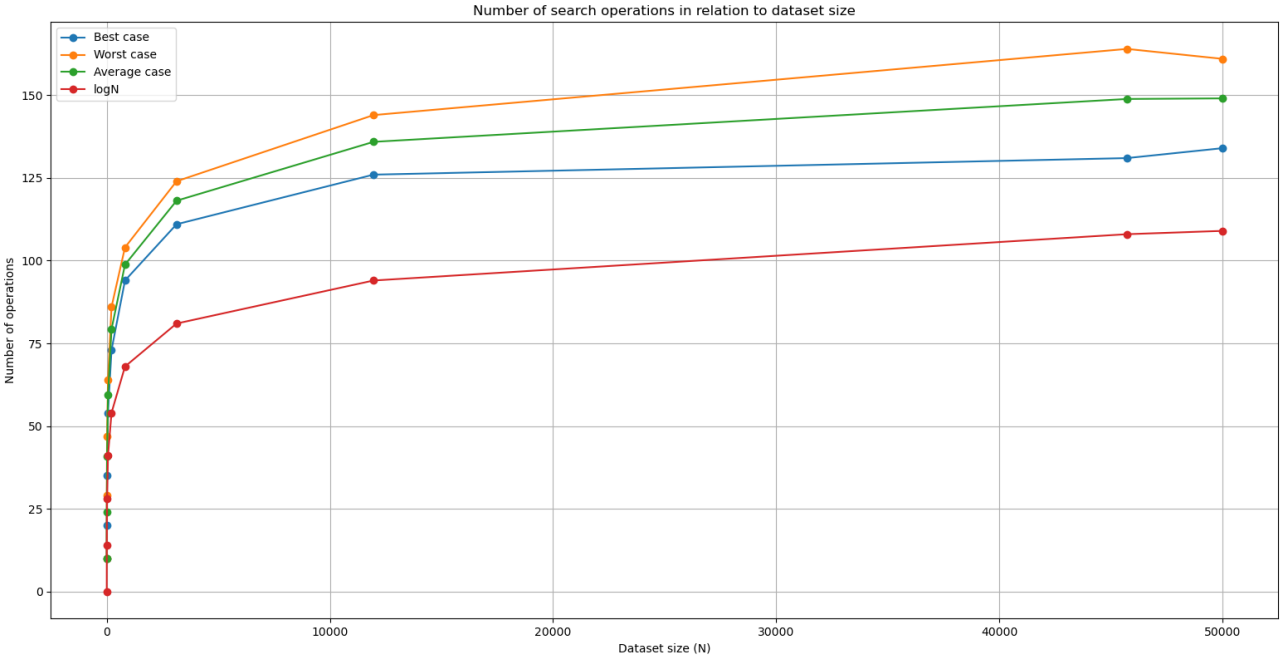
From these results we know that all our functions to add entries, queries and search for them all work as we included 5 queries that could be found and 5 that couldn't. From this we know our Knowledge Base works. Our instrumentation also works as the values are below our upper bound of $\log_2(10) \times 10 \approx 34$. We use base 2 as it is a binary tree.

Testing experiments

We will now test our full program using a randomised set of n as outlined under the experiments heading. Note that for this program we have fixed the size of our Queries10 and Entries10 to 10 entries. Therefore the only variance will come from our sample generics. Our python script automatically runs our program and stores the results of each run for each n in **Results.txt**. The values below and those in our assignment are after our program has been ran 100 times and aggregated the results. Note that it is not recommended to run this many times as it will take long. If you wish to run the program yourself it is recommended to run the test 5 times.

```
siyamthanda@ThatsMine:/mnt/c/Users/bheng/OneDrive/Desktop/csc2001f/csc2001f/Assignment2$ make run
python3 src/GenericsKBAVLTest.py
Enter the file name of where all entries are stored: GenericsKB.txt
Enter the file name for all queries: GenericsKB-queries.txt
How many times would you like to run this experiment:100
```





Show above are the graphs for the number of comparisons done for both search and insert for each value of value of n. For graph 1 it is important to note that it is the time taken to insert 10 entries into nodes of size n, not the time taken to populate a tree of size n. For graph 2 it is the time taken to search for 10 queries in a tree of size n. Looking at the general shape of both graphs we can confirm that the shape is indeed logarithmic. Even though our standard log n graph is below the other graphs we know that there exists a value M such that Mg(x) is greater than f(x). In our case our results are all multiples of log(n), therefore we can assert that our big-o for search and insert is O(log n). This makes sense as the constant rebalancing of an AVL tree means ensures that the maximum number if comparisons on the way to the last node log n.

From Results.txt we notice that all of our values are different but the still fall under our upper bound of $\log_2(n) \times 10$ even for the worst case. This tells us that the time complexity for insertions and search is indeed O(log n) as our graphs are multiples of log n.

Running the program:

Compile the program using **make**. Test part 1 by doing **make test.** In order to run the main program with the graphs type **make run**. The default file for entries is **GenericsKB.txt.** The default file for queries is **GenericsKB-queries.txt**. However you can run the program using any files that are formatted in the same manner. **GenericsKB-additional.txt** is used by the program to generate 10 random entries that are inserted into the entries tree, it should not be deleted. This program requires the downloading of matplotlib therefore doing a **pip install matplotlib** will ensure the graphs are all constructed properly.

Additonal features

This section includes all features that go above and beyond the scope of the originally prescribed assignment.

- Complete automation of the testing process using GenericsKBAVLTest.py.
- This script reads in you GenericsKB and creates multiple samples of different sizes n that are logarithmically spaced apart.
- It will also create a random Entries10 and Querie10 file that is kept standard for each run of the GenericsKbAVLApp.
- It will then run GenericsKbAVLApp using subprocess.run which allows you to input terminal commands using python.
- These files are then passed as args into the main file
- Also automates the process of doing multiple runs and storing an average of all runs for more accurate results
- Incorporated the use of Matplotlib so that graphs are created automatically based on the number of runs accuracy will increase.
- **make test** automatically tests if all the functions of the knowledge base work as intended.
- Results for all runs are stored in Results.txt
- Simple user interface so you can use any files you want for entries and queries as entries have at least 50 000 lines.